ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android

Andrea Possemato
EURECOM, France
andrea.possemato@gmail.com

Andrea Lanzi
Universita' degli Studi di Milano, Italy
andrea.lanzi@unimi.it

Simon Pak Ho Chung
Georgia Institute of Technology, USA
pchung34@mail.gatech.edu

Wenke Lee
Georgia Institute of Technology, USA
wenke.lee@gmail.com

Yanick Fratantonio
EURECOM, France
yanick.fratantonio@eurecom.fr

ABSTRACT

In the context of mobile-based user-interface (UI) attacks, the common belief is that *clickjacking* is a solved problem. On the contrary, this paper shows that clickjacking is still an open problem for mobile devices. In fact, all known academic and industry solutions are either not effective or not applicable in the real-world for backward compatibility reasons. This work shows that, as a consequence, even popular and sensitive apps like Google Play Store remain, to date, completely unprotected from clickjacking attacks.

After gathering insights into how apps use the user interface, this work performs a systematic exploration of the design space for an effective and practical protection against clickjacking attacks. We then use this exploration to guide the design of ClickShield, a new defensive mechanism. To address backward compatibility issues, our design allows for overlays to cover the screen, and we employ image analysis techniques to determine whether the user could be confused. We have implemented a prototype and we have tested it against ClickBench, a newly developed benchmark specifically tailored to stress-test clickjacking protection solutions. This dataset is constituted by 104 test cases, and it includes real-world and simulated benign and malicious examples that evaluate the system across a wide range of legitimate and attack scenarios. The results show that our system is able to address backward compatibility concerns, to detect all known attacks (including a never-seen-before real-world malware that was published after we have developed our solution), and it introduces a negligible overhead.

ACM Reference Format:
Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3243734.3243785

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243785

1 INTRODUCTION

Mobile devices are widespread and they have been subject to a significant corpus of research. One main area of works is about offensive research, which focuses on attacking these devices to highlight vulnerabilities. Within this context, a number of recent works has specifically focused on the mobile user-interface (UI). Many of these works have focused on the problem of *mobile phishing attacks* [4, 6, 9, 14, 20]. In such attacks, the user is tricked by a malicious app into inserting sensitive input (e.g., usernames, passwords) into a window that the malicious app controls. The core issue enabling these attacks is that users cannot understand whether they are interacting with a legitimate app (like a banking app) or a malicious one that is spoofing the legitimate UI.

Another class of attacks against user-interfaces (UI) is *clickjacking*, which is the focus of this paper. Such attacks work by creating an opaque overlay that completely covers a security-sensitive app (such as the Settings app): while the user believes she is interacting with an innocuous overlay, she is in fact interacting with the target app on the bottom (and she could unknowingly grant powerful permissions to a malicious app).

These attacks have been known for several years [1, 2, 17, 24, 26, 30, 31] and, in response, Google has implemented a security mechanism called "obscured flag." Such protection allows apps to detect whether, at the moment of the click, a sensitive widget button was covered by an overlay and, if that is the case, apps have a chance to refuse the click. Google adopted this security mechanism to protect the most security-sensitive of its Android apps, such as the Settings app.

However, a recent work called Cloak & Dagger (C&D from now on) showed how this defense mechanism can be bypassed [10]. The authors of this work developed a new attack, called *context-hiding attack*, which consists in covering the entire screen *except* the target button: In this way, the obscured flag protection does not trigger and, at the same time, the attacker is still able to *confuse* the user by hiding all the relevant security-sensitive context information.

In response to this attack, Google implemented an additional defensive mechanism: in recent versions of Android, when users browse to the accessibility service menu (the main target of the C&D work), all overlays drawn on top disappear. To the best of our knowledge, this hide overlays defense mechanism is sufficient to defeat clickjacking attacks (including C&D), mainly because the attacker does not have any possibility to confuse the user anymore. The common belief is thus that clickjacking is overall a solved

problem in mobile devices context. This paper, however, shows that this is not the case.

Clickjacking on mobile is an open problem. Clickjacking is not a one-off bug—it is caused by a design issue—and it is very challenging to prevent in the general case. In fact, the "hide overlays" defense from Google, while being effective, has two fundamental problems. First, we have identified several popular apps (with millions of users) whose core functionality—to act as a screen filter—specifically relies on creating persistent on-top fullscreen overlays. Thus, Google's defense cannot be widely adopted due to backward compatibility issues—it would in fact affect the user experience of all these apps with frequent interferences and flickering problems. Second, as we will show in this paper, Google deployed this fix to protect parts of the Settings app, but many other apps including Google own apps, such as the Google Play Store, and other popular third-party apps are left completely unprotected. We believe this is due to the backward compatibility concerns mentioned above. We note that the best defense available to third-party apps, the obscured flag, would also break the user experience for millions of users.

The design space for a practical defense. Solving this problem in the general case is very challenging since we need to design a protection system that is effective and, at same time, it does not break compatibility with existing apps. There are multiple proposed solutions, from both the academic and industry communities: We show that none of these are both effective (in preventing attacks) and practical (especially when applied in all new existing attack scenarios described in this paper).

To prevent the design of another problematic solution, in this paper we first gather insights on how apps use the user interface, and we then systematically explore the design space by drawing a number of observations that, independently from a given proposed solution, we believe must all be taken into account when designing a system that is both effective and practical. This exploration guided us to the design of a new protection mechanism, dubbed ClickShield. Our defense differs from existing ones because it tackles the problem of clickjacking at its root: the possibility of deceiving the user. We thus do not focus on the many technical ways single overlays can be created—too many to be properly enumerated—and we focus instead of the *net effect* that these overlays have on what the user actually sees on the screen. To this end, we devised efficient techniques based on image analysis to answer questions such as *is what the user seeing different than what the target app would have liked to display? Was the view from the target app modified? If yes, was it modified in a uniform way so that the full (potentially security-related) context is still available for the user?*

We have implemented a prototype of this system, and we have tested it against ClickBench, a newly developed benchmark specifically tailored to stress-test clickjacking protection solutions. This dataset is constituted by 104 test cases, and it includes real-world and simulated benign and malicious examples that evaluate the system across a wide range of legitimate and attack scenarios. We note that some of these test cases have been developed specifically to evade our own system and that we included in the benchmark even a never-seen-before real-world malware sample that was made public after we had finalized our prototype. Nonetheless, ClickShield is able to detect all attack scenarios without being affected by backward compatibility concerns. Moreover, our proposal has a negligible performance impact, and it is thus suitable for adoption on mobile devices. We believe ClickShield to be the first practical approach that has the potential to fully eradicate clickjacking on Android.

In summary, this paper makes the following contributions:

- We highlight how clickjacking on mobile devices is still an open problem and how the attack surface is much wider than what previously thought.
- We show how current defense mechanisms fall short and we discuss the main challenge Google is facing: backward compatibility issues, which would break core functionality of popular apps used by millions of users.
- We gather insights on how apps use the user interface and we systematically explore the design space for an effective and practical defense mechanism. We build on these insights to design ClickShield, a novel defense mechanism for mobile clickjacking.
- We evaluate ClickShield against ClickBench, the first benchmark dataset for clickjacking solutions. We show that our system is effective at stopping the threat of clickjacking and it addresses backward compatibility concerns.

To ease the reproducibility of this work, we will publicly release our prototype and our benchmark dataset.

2 BACKGROUND ON ANDROID UI

In Android, third-party apps having the SYSTEM_ALERT_WINDOW permission have the ability to create arbitrary windows, also known as *overlays*, that are rendered on top of the current activity. For apps hosted on the official Google Play Store, this permission is automatically granted, without the user being notified about it. Apps have complete control over the overlays they create. In particular, they can control their size and position, and whether they are opaque or (semi-)transparent. Apps can also create overlays that are either *clickable* (i.e., when the user clicks on them, the overlay will capture the click) or *passthrough* (i.e., the click is not captured by the overlay and it is passed to the overlay or activity beneath it). Starting from Android 8.0, apps are forbidden to draw overlays on top of the lock screen, the status bar, and the navigation bar. While these new constraints are an effective protection against ransomware (because it does not have a chance to completely lock the device), they have no impact against clickjacking attacks because overlaying these sensitive UI components is not relevant.

3 CLICKJACKING ON ANDROID

This section discusses current techniques to perform clickjacking on Android and the security mechanisms in place to prevent them.

Traditional clickjacking attack. The essence of a clickjacking attack is about confusing the user and luring her to perform a "click" action so that the attacker achieves her malicious goals (e.g., additional permissions are granted). Traditional clickjacking attacks, introduced in [17], consist in the following steps: 1) The attacker creates an overlay that is fullscreen, opaque, and passthrough; 2)

Unbeknownst to the user, the attacker spawns the victim app (e.g., the Android Settings app) below the opaque overlay; 3) The malicious app lures the user to click on a specific point on the screen; and 4) The click passes through the opaque overlay and reaches a security-sensitive button beneath the malicious app, at which point the attack is completed.

While the steps above focus on hijacking one single click, the recent Cloak & Dagger work [10] showed how this technique can be easily extended to a multi-click scenario: by using a combination of flags when creating the overlays, the attacker can create a side-channel to infer that the user has just clicked where she was supposed to click; upon the reception of this "signal," it can then modify the on-top overlays to lure the user to click on the next button. Of course, the higher the number of clicks required, the less practical the attack is. However, the authors of C&D work showed through a user study that even an attack requiring three clicks is very practical.

Obscured flag defense. To protect from these threats, Google implemented a mechanism to allow apps (both Google's own and third-party ones) to protect themselves. This mechanism, called "obscured flag," works by signaling (via a boolean flag) to `Button` widgets that, when the click was performed, an overlay was covering (or "obscuring") it, independently from whether the on-top overlay is opaque or transparent. This is how the Android framework signals to an app the possibility of an on-going clickjacking attack. Google adopted this mechanism to protect its most sensitive Android apps, such as the Settings app.

Context-hiding attack. Although the obscured flag mechanism raises the bar for attacks, it was recently discovered to be easily bypassable, with a technique called "context-hiding attack" [10]. The key observation behind this technique is that as long as an attacker can hide the real, security-sensitive context surrounding a generic OK button, it is easy to lure the user to click on it. Thus, by covering the entire screen *except* the target OK button, the obscured flag defense can be bypassed.

Hide overlays defense. Finally, to counter the threat of context-hiding attack, Google implemented a new defense mechanism to prevent it: in modern versions of Android (from Android 7.1.2), when the user browses to the accessibility service menu or permission settings, *all overlays are hidden*, thus removing the possibility for the user to be confused (and for malware to mount an attack).

Current Limitations. The "Hide overlay" defense is very effective: Since all the on-top overlays are hidden, there is no chance for the attacker to confuse the user, and, to the best of our knowledge, clickjacking is thus prevented. However, this mechanism is very aggressive and, as we will describe in Section 6, it has two limitations. First, it is too powerful to be made available to third-party apps, which thus remain unprotected. Second, it creates a number of severe backward compatibility issues, which would break the main functionality of apps installed by millions of users. The obscured flag mechanism is affected by similar backward compatibility issues. It is in fact not uncommon to read about users puzzled by usability problems due to these mechanisms [27]. This aspect pushed Google to adopt this security mechanism only to protect the most sensitive parts of the Android system (such as the permission granting popups), leaving many sensitive Google-owned apps (such as the Google Play Store app) completely unprotected.

4 NEW ATTACK SCENARIOS

This section discusses known and several previously unknown clickjacking attack scenarios. The feasibility of these attacks has been tested on a fully updated Nexus 5X running the latest version of Android (8.0) available at the time of writing. For the sake of completeness, we include in this discussion previously known examples that are now prevented by currently deployed security mechanisms. The list of attacks, their feasibility, and their novelty are systematize in Table 2, in the Appendix.

Previous works have shown how clickjacking can be used to lure the user to unknowingly grant additional permissions (e.g., the "location" permission) or even to enable accessibility service shown to be enough to fully compromise the device [10]. Google has now fixed these attacks by implementing the "hide overlays" defense mechanism.

Another related work [29] has shown how clickjacking can be used to bypass several permissions, such as capturing images and videos (Target App (TA): Camera app), getting access to contacts (TA: Contact app), record sound (TA: SoundRecorder), send text messages (TA: Messaging app), or even installing and uninstalling third-party apps (TA: Package Installer). Among these, attacks against the Package Installer are now protected via the obscured flag mechanism. However, even this last case is still vulnerable to the context-hiding attack. According to our tests, all the other attack venues are still practical on the latest version of Android. This is particularly worrisome when considering that this related work has been published two years ago, in mid 2016.

In this paper we explore additional attack scenarios, and our findings are alarming.

Google Play Store app. We found that even Google's own Play Store app is completely vulnerable even to traditional clickjacking attacks. In fact, it is not even protected by the obscured flag mechanism, making its exploitation trivial. This is problematic since this app has the capability of installing, uninstalling, and opening arbitrary apps installed from the Play Store.

An attacker can cause the Play Store app to open and "browse" to an attacker-chosen app by sending an ACTION_VIEW Intent and a URI with the `market://` scheme (e.g., `market://details?id=malicious.com`). If the app is not installed, the Play Store will show, in its first activity, the "Install App" button, making it possible to install an arbitrary app from the Play Store by hijacking *one* click. After the app is installed, the malicious app can send the same Intent: this time, since the app is already installed, the Play Store app will show an "open app" button. Thus, by hijacking two clicks only, an attacker can install and open an arbitrary app from the Play Store. To make things worse, we have found that if the attacker-chosen app targets an old Android SDK, the full list of permission is shown to the user at install-time (in contrast to the current grant at run-time permission model). Unfortunately, we have found that the "OK" button to confirm these permissions is also vulnerable to clickjacking. In summary, by hijacking three clicks, an attacker can lure the user to install an arbitrary app from the store with arbitrary permissions.

Chrome Browser. The mobile version of Chrome Browser is vulnerable as well. An attacker can simply open an arbitrary webpage by sending an ACTION_VIEW specifying the target webpage's URL as data. If the user is logged in to the target site, the attacker can use clickjacking to implement traditional web clickjacking attacks (e.g., to click on Facebook's likes), bypassing all modern web-related defense mechanisms (such as frame busting).

Gmail. Google's Gmail app is vulnerable. By using an ACTION_SEND Intent, by setting `com.google.android.gm` as target package, and by setting the EXTRA_EMAIL, EXTRA_SUBJECT, and EXTRA_TEXT extra fields, an attacker can spawn the Gmail app with a pre-filled email (including the To:, Subject:, and content of the email). The attacker can then hijack a click to the "Send" button, with the net effect of being able to send emails on behalf of the victim, which could be useful to mount social engineering and targeted attacks.

WhatsApp and Signal. These applications are very popular among the Instant Messaging (IM) apps: WhatsApp is one of the most used messaging application across Android users while Signal is considered the de-facto standard for secure messaging. Both allow the user to perform end-to-end encrypted communications. Unfortunately, these sensitive apps are also vulnerable to clickjacking.

For what concerns WhatsApp, an attacker can send one crafted Intent so to pre-fill the content and the recipient of a message to be sent: by hijacking just one click, such message will be sent on behalf of the victim. This technique can be abused to leak the victim's telephone number by sending a message to an attacker-controlled number; the attacker could also use this attack vector to impersonate the victim to perform social engineering attacks or spam-related activities.

Signal is vulnerable to this attack as well, but only if it is configured to be the default app for handling SMS: in case it is not, Signal does not allow the creation of a pre-filled message with an arbitrary, attacker-controlled recipient, making the one-click attack not possible. We note, however, that a clickjacking-based attack against Signal is possible even in its default IM mode, but it becomes more complicated. In particular, a malicious app could ask the WRITE_CONTACT permission to add the attacker's phone number to the victim's contact list multiple times, and it could use a specially crafted name so to reach the top of the contact list (like in a spraying attack): then, by hijacking two clicks—the first one to select the attacker-controlled recipient (which is now on top) and the second one to actually send the message—the attacker can once again leak the victim's telephone number. The attacker could then clean up the contact list just after the attack is over.

Google Authenticator. Google Authenticator is also vulnerable to clickjacking, but in a different way. For this app, there is no sensitive button to be clicked, but it contains sensitive information, such as two-factor authentication tokens. We have developed a clickjacking-based technique through which the attacker can leak this data. By luring the user to perform a "long click" action on one of the tokens, *the Google Auth app will copy this token to the clipboard, which is freely accessible by any third-party app without requesting any additional permission.* We were able to quickly write a prototype that, by just hijacking one click, obtains the relevant token from the clipboard. To the best of our knowledge, this is first known example of combining hijacking of long clicks and leaks via clipboard. We note that this technique is generic, and that we focused on Google Auth only as an explanatory example. In fact, our tests show it is also possible, for example, to attack the Google Drive app and lure the user to click on "share by link" for a given item or folder, after which the item is shared and the link is copied to the clipboard (and thus leaked to the attacker).

Facebook and Twitter. These apps are completely unprotected, and it is thus easy for an attacker to "like" or share messages and tweets, with techniques similar to what described above. For example, it is possible to spawn the Twitter app to show a specific tweet (URLs that contain twitter.com are treated in a special way and delivered to the Twitter app), and it is thus trivial to perform like-jacking or similar attacks.

Lookout Mobile Security. As a last representative example, we investigated how the leading anti-virus app for mobile is resilient to these attacks. To our surprise, we found that all widgets were vulnerable to clickjacking: by hijacking three clicks, it is possible to silently disable the security checks.

Discussion. The intent behind this section is to highlight the extent and wide attack surface that even fully updated devices are subject to. The practicality of these attacks varies depending on the number of clicks to be hijacked. However, we note that the most complex example above is attacking Signal, which requires four clicks, but that previous work has shown through a user study that these multi-step clickjacking attacks are very practical [10].

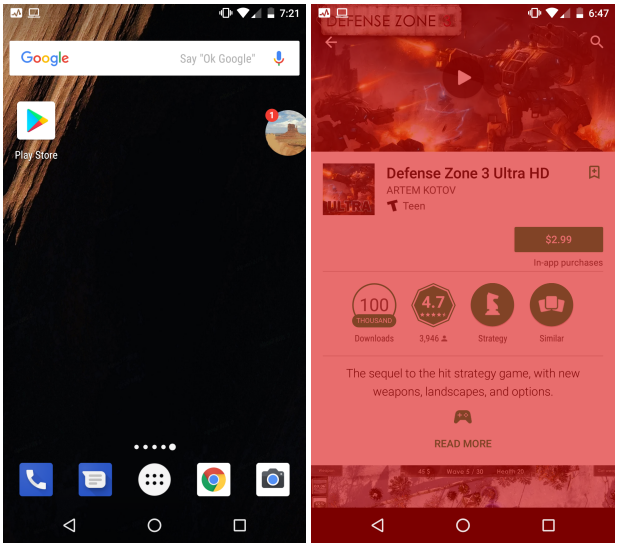
This is a list of interesting findings, but we would like to stress that it is far from being complete. However, we find it worrisome that, among the apps we have tested, *we have not found a single one that was protected by at least the obscured flag mechanism.* On the one hand, it is surprising that all these apps are still vulnerable, especially since clickjacking for mobile has been known for several years. On the other hand, we believe that this lack of protection is not due to simple oversights, but it is due to fear of public outcry caused by these backward compatibility concerns [27]. For example, we have notified Google about the attacks against the Play Store in August 2017, but, unfortunately, they are still practical.

5 HOW APPS USE THE USER INTERFACE

We have conducted a survey to determine how real-world apps use the user interface. We have tailored our survey to study two specific aspects: 1) how benign apps convey to the user the needed contextual information concerning sensitive and security-relevant actions; 2) how benign apps use the "draw on top" permission, and which functionality they aim at implementing.

5.1 Conveying Contextual Information

We wanted to study how apps make use of the user interface to convey relevant contextual information for a user to take informed decision. Note that with the term *contextual information* we do not only refer to security-related information displayed, for example, in a popup: with it, we consider all information that the user needs to be aware of to determine what the effect of one of her click on a button would be (e.g., a click on the bottom-right part of the Gmail app triggers a "send email" action).



(a) (b)

Figure 1: (a) An example of "widget" implemented as an on-top overlay by Facebook Messenger. (b) Twilight app in action, an example of screen filter. The red shade is implemented as a persistent, on-top, semi-transparent overlay.

To this end, we have compiled a cumulative list of 67 sensitive views used by different types of apps. This list includes views from the Android settings app (and its sub-menus, 11 entries in total), Google-owned apps (Gmail, Google Drive, etc., 11 views), and several representative sensitive views belonging to banking apps (15), social networks like Twitter and Facebook (8), messaging apps including Telegram, Signal and WhatsApp (8), and security apps such as mobile antivirus (14). In all cases, *the relevant contextual information is prominently shown in the center*. We note that none of the known and just-presented attacks discussed above would be possible without covering the central part of the screen. Thus, we conclude that, at least for the apps we have inspected, as long as the central portion of the screen is not covered, clickjacking attacks are not possible.

5.2 How and Why Apps Create Overlays

One of the concerns when developing a security mechanisms relates to backward compatibility. As we aim to design a mechanism that is not affected by these concerns, we have performed a survey over a number of real-world apps to study how they use the "draw on top" permission, and which purpose they want to achieve by drawing overlays. We built three different datasets, which aim at covering potentially problematic categories of apps (for what concerns clickjacking protection mechanisms).

The first dataset is composed by popular apps hosted on the official Google Play Store. This dataset is constituted by 454 apps, and they all require the SYSTEM_ALERT_WINDOW permission. This list was kindly provided by the authors of Cloak & Dagger [10], and it was obtained by filtering for apps requiring the permission from an initial dataset of 4,455 top apps on the Play Store crawled across different categories. We have considered a subset of (randomly selected) 305 apps for closer inspection. These apps span a number of categories, including screen filters, messaging, audio & video players, photo & video editing, custom launchers, VPN & networking, productivity & utilities (e.g., status indicators), antivirus, and screen lockers.

The first dataset is already quite significant in size, but we wanted to include in our analysis 1) apps from different sources and 2) apps that could be particularly problematic for a defense mechanism. To this end, we created a second dataset with apps taken from the F-Droid open source apps repository [7]: we have randomly selected the top 20 entries on Google when searching for the "android.permission.SYSTEM_ALERT_WINDOW" permission used in webpages belonging to F-Droid apps. Among these, only 15 were real apps (the remaining ones were toy samples) and they, once again, belong to a number of different categories, such as messaging, VPN & networking, productivity & utilities, audio & video players, photo & video editing, and custom launchers. Moreover, we have also created a third dataset constituted exclusively by screen filter apps. We chose to focus on this category because their key functionality is well-known to create backward compatibility issues with current protections against clickjacking attacks [27]. In fact, this category of apps relies on the creation of persistent fullscreen, passthrough, on-top overlays: these overlays are all detected as problematic by the obscured flag mechanism. For this dataset, we have selected the 10 screen filter apps with the highest number of installations from the Play Store. We report these apps in Table 1 in Appendix B.

We have selected for a throughout manual inspection a number of samples (between 3 and 5) for each of the categories that were covered in the first dataset, and all the samples in the second and third dataset, for a total of 60 samples. The remainder of this section discusses the gathered insights, grouped by the different functionality these apps aim at implementing.

Widgets at the margin. One very frequent use-case for the apps in our dataset is to create overlays with the purpose of drawing persistent widgets. Depending on the app category, these widgets are used to display a number of information, to act as shortcuts, or to attract attention from the user. Figure 1a shows a very popular example, Facebook Messenger, which draws a rounded overlay to notify the user of a new message. Other categories of apps that create this kind of widgets are audio & video players (to show which song is playing, see Figure 7a in Appendix C), VPN & networking (to show the status of the connectivity, strength of the signal, and similar information, Figure 7b), custom launchers (that draw side widgets with shortcuts to various apps, Figure 9a and 9b), status indicators (e.g., battery level, Figure 10a), and productivity apps (that create shortcuts to documents, notes, and calendars, Figure 10b).

We note that, in all cases, these widgets are drawn opaque, clickable, and they are place at the margin of the screen. This is expected, as users would likely be annoyed by on-top opaque overlays in the middle of the screen. We also note that these apps do not conflict with the obscured flag defense. In fact, this defense kicks in only when overlays cover the security-sensitive button itself. However, the "hide overlays" defense could cause problems, because the users

would see these overlays flickering whenever the user clicks on a security-sensitive button.

Screen filters. The main goal of these apps is to allow the user to setup screen filters, so to change the "tone" of the screen color. Many of these apps offer this feature to help users addressing sleep-related problems. To quote the app description of Twilight [28], one of the most popular apps of this category, "exposure to blue light before sleep may distort the user's natural (circadian) rhythm and cause inability to fall asleep." Thus, these apps change the color of the screen to filter these blue-related components. Figure 1b shows an example: as the reader can note, the screen's color is shifted towards a red component (this is a side effect of attempting to remove the blue components).

Table 1 (in the Appendix) lists a number of these apps, together with their number of installations. These apps appear to be very popular: Even the most conservative estimation of the users of these apps would conclude we are in the order of, at least, tens of millions of users. We manually tested them all, and we confirmed they all work by creating a persistent, semi-transparent, fullscreen overlay on top of every other app.

Unfortunately, these apps break both the obscured flag and the hide overlays defenses, making them inapplicable, if not in very specific highly sensitive cases, such as the Settings app. In fact, when using them, the user cannot click on any security-sensitive button when these overlays are up.

Show important notifications. We have found apps belonging to the antivirus category that create overlays to show "urgent" notifications, e.g., a malware was identified. These overlays are created as opaque and clickable (and usually show some kind of information), and they are thus created in the central part of the screen.

Photo & video editing. We have found one sample that embeds a camera view (the app requires the "camera" permission), on top of which a few overlays are superimposed. The main purpose of these overlays is to implement menu shortcuts and display current information such as GPS position. We note that these overlays are only drawn when the app is open—they disappear as soon as the app is closed.

Content preview. One of the samples belonging to the audio & video category (from the F-Droid dataset) creates an opaque, clickable overlay in the center of the screen with the main goal of showing a preview of the video that is about to be played.

Screen lockers. Apps belonging to the screen lockers category create overlays with the only intent of blocking the screen. Thus, these overlays are created fullscreen, opaque, and, of course, clickable. Once the lock screen is de-activated, the overlay disappears. Note that starting from Android 7.0, these apps are not useful anymore: a user can easily access the notification bar and disable these overlays, thus making screen lockers easily bypassable.

Discussion. Among the various "benign" use cases that we have mentioned above, some of them may potentially interfere with a clickjacking defense. In particular, we refer to all those cases that, by design, rely on somehow persistent overlays, i.e., overlays that do not appear as a one-off notification, but they are specifically designed to remain on the screen for an extended amount of time. The main two classes are screen filters and apps that display persistent widgets: these are very popular (100+ million users), and they thus require to be taken in consideration.

However, the remaining use cases, such as apps that show notifications, photo & video editing apps, or screen lockers are not problematic: the overlays they create are either not persistent, or they are created only when the user is interacting with the app generating them (e.g., photo & video editing). In other words, in a benign scenario where the user interacts with a sensitive app, these overlays would not be displayed in the first place.

6 DESIGNING A DEFENSE MECHANISM

The design space for an effective mechanism to protect from clickjacking is large. To make it worse, as we will see, there are a number of solutions that seem to work [13, 29], but then turn out to not cover all malicious cases, or introduce significant backward compatibility issues (e.g., hide overlays). To this end, we opted for a systematic exploration of the design space and we draw a number of observations that, independently from a given proposed solution, we believe must all be taken into account when designing a system that is both effective and practical.

6.1 Prerequisites for Clickjacking

We aim at designing our solution to target the fundamental properties of clickjacking. Here we briefly systematize the two prerequisites that *every* variant of clickjacking attack needs to satisfy.

P1: A click needs to reach the vulnerable app. By the very definition of clickjacking, the attacker needs to lure the user into clicking on a widget owned by a vulnerable, security-sensitive app. In other words, no clickjacking attack can be successful if the click does not land on the target app.

P2: Confusing the user is necessary, by definition. In order to lure the user into clicking the target button, the attacker must hide or tamper with the current contextual information, even if in part. In fact, if the context were *fully* available, the user would have all the needed information to notice the (negative) impact that her click may have, and she would never knowingly perform such action. Note that when we talk about hiding the context, we do not necessarily refer to malicious apps that fully (or prominently) cover the main portion of the screen; instead, we also refer to more subtle cases where few pixels on the screen are changed so that, for example, a purchase price is slightly altered.

6.2 Exploring the Design Space

In this section we systematically explore the design space of possible defense mechanisms. We organize the discussion by focusing on *six* complementary aspects.

6.2.1 Overlay location

O1: Overlays at the margin are not problematic. As discussed in Section 5.1, we have studied how apps convey contextual information to their users, and, we found that, in all cases we have analyzed, such information is prominently displayed in the central part of the screen. Thus, one key observation is that while it is important to make sure that the central part of the screen remains visible to the user, overlays displayed at the margin of the screen

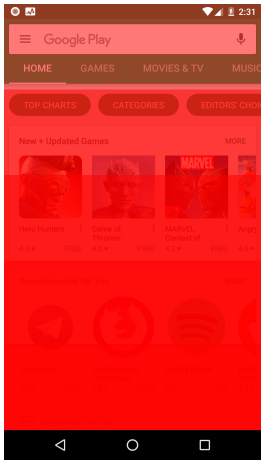


Figure 2: This figure shows how multiple overlapping semi-transparent overlays are rendered as opaque. It has been obtained by overlapping one, two, three, and four identical and semi-transparent overlays.

can be considered as not problematic. In fact, none of all the attacks discussed in the previous section would be possible if the central part of the screen would not be completely covered. Similarly, we have not identified any single practical scenario where an attacker could perform clickjacking attack by just using a Facebook Messenger-like widget on the margin of the screen.

O2: Overlays covering the center of the screen must be allowed. Yet, as discussed in Section 5.2, several popular apps require to implement for their main (often single) functionality the creation of persistent, on-top overlays. This is the case, for example, of screen filter apps. Thus, to avoid backward compatibility issues, a practical defense mechanism needs to allow this to happen.

6.2.2 *Clickable vs. passthrough*

O3: Both clickable and passthrough overlays could be used to mount attacks. Traditional clickjacking attacks require the usage of passthrough overlays, so that the click of the user would reach the victim app beneath the on-top overlay. One would thus be tempted to consider the usage of passthrough overlays as a necessary condition for clickjacking, and thus use this as a detection "feature." However, *the context-hiding attack can be performed with either clickable or passthrough overlays*. In fact, in this scenario, it is even more practical to use clickable overlays, so that the attacker can catch all clicks that do not land exactly where the attacker wants. Unfortunately, this is an assumption used in a recent proposal by Wu et al. [29], which can thus be bypassed.

6.2.3 *Opaqueness vs. transparency*

O4: Both opaque and transparent overlays could be used to mount attacks. Similarly to the point above, one may be tempted to treat opaque overlays as (possibly) malicious, but to consider semi-transparent overlays as benign. This is another assumption used by Wu et al. [29], and we think it offers another venue for bypass. In fact, we wrote a proof-of-concept that shows how *a malicious app could easily create a number of overlapping semi-transparent overlays that, when rendered all together, behave as an opaque overlay* (due to how the rendering procedure works). We show this effect in Figure 2, where it is possible to see how the main part of the screen is completely covered, even if all the overlays used in this example are in fact semi-transparent. Thus, one should refrain from considering overlays independently: what matters is the *net effect* of all these overlays.

6.2.4 *Content analysis*

O5: Machine learning techniques should be avoided. A number of works have shown how even the most advanced machine learning techniques are vulnerable to sophisticated adversarial attacks, especially for algorithm working in the image domain [5, 18]. Thus, we believe that it is preferable to avoid defense solutions based on machine learning (to perform image similarity tests, for example), especially since, in this scenario, a malicious third-party app would have full control over the pixels displayed to the user. We thus exclude techniques that rely on AI-based image recognition, such as image similarity analysis or Optical Character Recognition (OCR) techniques.

O6: Apps can collude. One option to detect malicious overlays is to detect the overlays generated by a given app, and analyze them for malicious signs. However, *it is possible for malicious apps to collude and create overlays that are benign when considered separately, but malicious when rendered together*. Once again, we believe it is important to focus on the net result of all displayed overlays.

6.2.5 *When should the defense come into play?*

O7: Detection only at clicks on sensitive widgets. A necessary prerequisite for a clickjacking attack is that the user clicks on a security-sensitive button (P1). This implies that as long as no click reaches such sensitive widgets, no attack can happen. Thus, it is possible to run the detection algorithm only in these cases, without the risk of having attacks going unnoticed.

6.2.6 *How to implement such defense mechanism?*

O8: Limited information should be exposed. The defense mechanism should not expose, directly or indirectly, sensitive information to third-party apps. Third-party apps should be able to determine whether the user clicks on their button while being fully aware of the action she is authorizing. However, they should not be able to infer information such as "which overlay has the user clicked on," or "which app was covering it." Otherwise, a malicious app could use this security mechanism as a side-channel to infer what the user is typing (see keystroke recording via obscured flag in [10]), and which and when a user opened a given app (such as a banking app), thus facilitating the practical development of phishing attacks [4, 6, 20].

O9: No special permission should be granted to third-party apps. The defense mechanism should not allow third-party apps to affect the user experience of other applications. This is the reason why we believe that the "hide overlays" defense mechanism implemented by Google is not a good candidate to be granted to third-party apps as well—it would be a too powerful mechanism, and likely to be abused.

O10: Framework modifications are required. There are a number of proposals, as we will discuss in Section 10, that attempt to prevent clickjacking and similar attacks by providing a user-level library, without requiring framework modifications. We argue that framework modifications are necessary if we were to determine not only that an overlay is on top of a security-sensitive button (this is information is exposed via the obscured flag mechanism), but also to determine *whether the contextual information is obscured as well.* Having access to this last information is critical to fully prevent clickjacking, but it is currently not exposed by the Android framework. We thus believe that any proposal that does not require modifications to the framework (or privileged access) cannot fully eradicate this threat.

7 CLICKSHIELD

This section describes the defense mechanism we have designed for this work. The proposed system, called ClickShield, is envisioned as a modification to the Android framework. Its design is built upon the observations discussed in the previous section. In this section we will refer to these observations with the O*n* notation, where *n* identifies which observation we are referring to.

7.1 Overview

The vision. Our system's main goal is to implement a principled defense for clickjacking attacks. To lower the barrier for adoption, the integration scenario we envision for this work is the same as the currently existing obscured flag defense mechanism: the developer indicates to the Android framework that a given Button widget is associated with security-sensitive operations, and thus needs to be protected. We refer to these buttons as *sensitive buttons.* One of the necessary condition for any clickjacking attack is that a click needs to reach a sensitive button. Thus, the defense mechanism presented below enters in action *only* when a click on a sensitive button is detected (see prerequisite P1).

Security assessment. Once a click is detected, the system checks for the presence of overlays. If no overlays are detected, then no attack can happen, and the system quits the analysis. If at least one overlay is detected, the system checks *where* this overlay is located. We know that the relevant security context is shown to the user in the middle of the screen (see Section 5.1), and overlays located at the margins of the screen can then be treated as innocuous (O1). For example, overlays generated by Facebook Messenger (see Figure 1a) fall in this category, and they thus do not raise unnecessary warnings. However, of course, overlays could also be displayed in the middle part of the screen. On the one hand, these overlays could attempt to alter/hide the current context; on the other hand, it is not practical to disallow such overlays, as it would generate too many redundant warnings (O2).

The key idea. While the implementation of simple checks such as presence and location of overlays is trivial, the key technical challenge we focus in this work is to *determine whether the overlay(s) covering the center of the screen are attempting or have the potential to deceive the user.* One would be tempted to analyze each of these overlays separately, and determine whether they are semi-transparent and uniform. However, as noted above, it would be easy to bypass this technique (O4 and O6). Our idea is to focus on analyzing and comparing *what the user sees* against *what the user would have seen if no overlay were rendered.* This idea has the advantage of abstracting away *how* the final rendered image that the user sees is generated: it is in fact independent from whether the overlays are clickable or passthrough (O3), whether they are opaque or transparent (O4), and it is independent by their number and from which app or apps have generated them (O6). This idea is implemented in a new image analysis technique we called *deblending*, described in details later in this section.

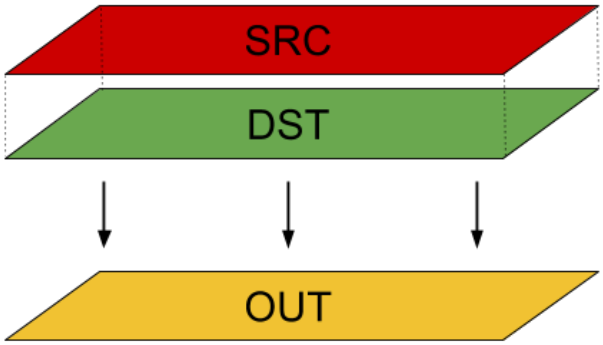


Figure 3: Representation of the Alpha Blending mechanism.

Protection. The output of the above analysis steps can have two outcomes: either the system returns "benign" or it returns "suspicious." In the first case, no step is taken and everything proceeds as expected—the user will not even realize that a security check was ever performed. If the outcome is "suspicious," however, the system takes additional steps to ensure that the user is knowingly clicking on the sensitive button. To this end, the click is not delivered to the target app, all overlays are temporarily hidden, a warning is displayed, and the user is asked to click again, if that was intended. Additionally, one could envision a system that allows a user to report the offending app, in case this was a true positive. We also note that, in our deployment scenario, our system would be implemented directly within the Android framework itself, and not in the third-party app. This is in stark contrast with the implementation of the obscured flag mechanism, in which the developer of the benign app needs to write the error handler, thus creating possibility of bugs and increasing the friction for adoption.

7.2 Deblending

We now discuss the core technical part of our work, the *deblending* analysis step. Before we do that, we will cover the technical background needed to understand the details.

7.2.1 *Alpha Blending*

Android apps can create overlays that will be displayed "on top" of the current application. Each overlay can be arbitrarily controlled by the generating app. For each of its pixels, the app can set the Red, Green, and Blue components, as well as the Alpha component, which defines the "transparency level" of the pixel. Thus, each pixel is defined by a quadruple of 8-bit numbers, RGBA. These overlays can be fully opaque (like Facebook Messenger overlay in Figure 1a),

or semi-transparent (like Twilight app, shown in Figure 1b). Apps like Twilight achieve this by specifying, for each pixel, an alpha value lower than the maximum (i.e., 255).

Overlays are rendered together by an Android framework component called `SurfaceFlinger`, which implements a technique known as "Alpha Blending." Alpha blending defines how the different overlays are "mixed up" according to their RGB components and their alpha values. Consider Figure 3. The application the user is interacting with is represented by DST, while the overlay (one or more) is represented by SRC. Given the pixels for DST and SRC, alpha blending determines, according to a formula, what the user will see on the screen, represented by OUT. For example, this is how the red component of OUT's pixel at position (x, y) is computed:

$$OUT^{red}_{x,y} = round(SRC^{red}_{x,y} * \alpha + DST^{red}_{x,y} * (1 - \alpha)),$$

where DST and SRC (integer values between 0 and 255) indicate the red components of DST's and SRC's pixels at position (x, y), α indicates the alpha value of SRC's pixel at position (x, y), and $round$ indicates the rounding to nearest integer operation. Note that in this formula, α is a float value ranging from zero to one. If $\alpha = 1$, SRC is fully opaque: as expected, $OUT = SRC$; if $\alpha = 0$, SRC is fully transparent, thus $OUT = DST$.

7.2.2 The Key Idea

In our scenario, DST represents the target app, SRC is what is drawn on top of it, and OUT is what the user actually sees. In Figure 3, SRC is shown as a single overlay. However, we note that SRC can be interpreted as the "sum" of all drawn overlays, rendered against DST, which then generate the final image OUT. In other words, SRC represents the "delta" between *what the target application expects the user would see assuming no overlays are displayed* (DST) and *what the user actually sees* (OUT). We note that this abstraction captures all cases where one or more overlays are displayed, regardless of how these overlays are created (clickable or passthrough, opaque or semi-transparent), and regardless of which and how many apps are creating them. Thus, this captures complex scenarios such as, for example, a malicious app taking advantage of overlays created by benign apps, or multiple colluding malicious apps.

Within the context of this setup, the key idea is to first extract DST and OUT, and to use them to compute a candidate for SRC (note that, in fact, SRC is not directly available to the system without additional rendering operations, which would constitute an invasive change and incur in performance overhead). We then analyze SRC to determine whether the overall impact of all the overlays that cover DST can be deemed as benign or suspicious. We consider the computed SRC as benign if and only if it is, in fact, a semi-transparent and uniform overlay. The uniformity constraint implies that DST has been altered in a "uniform" way and no parts of the screen are modified differently than others; while the semi-transparency constraint implies that DST is indeed still visible. We note that this "uniformity" property is intentionally a *global* property. Take the benign example of Twilight: its overlay is modifying each and every pixel on the screen, but it does so in an uniform way, and it is thus considered as a benign case. The rest of this section discusses these steps in detail.

7.2.3 Extraction of DST and OUT

To compute SRC, we first need to extract DST and OUT. DST is how the user would see the target app without any overlay. In other words, it represents the "pure" version of what the target app expects the user to see. OUT, instead, is what the user actually sees. To extract these images, we have modified the Android framework's `SurfaceFlinger` component. Our modification exposes a new API that allows us to extract both DST and OUT. In particular, OUT is simply what is displayed to the user; instead, DST is obtained by filtering out all the layers that are not generated by the target app (the one that received the click). Since the rendering is performed by processing layers from the lowest to the highest (for obvious reasons), DST can be easily obtained by dumping the current state before overlays above it are processed. The same argument holds for OUT: the renderer would have needed to compute it regardless of our defense system. Thus, from the conceptual point of view, no extra rendering steps are needs for this task. The subsequent analysis steps rely on having access to the raw values of pixels for DST and OUT.

7.2.4 Computation of SRC

Assuming we have access to the raw data for DST and OUT, the goal is now to compute SRC. We want to find SRC such that OUT can be "explained" starting from DST. One may be tempted to calculate SRC pixel by pixel. However, this is not possible. Recall that within the context of the alpha blending formula, DST and OUT are known, but SRC is not. Thus, given the RGB values of DST and OUT for a single pixel, we have four variables (SRC's values of RGBA quadruple), but only three equations (one for each RGB component of OUT): the system of equations is *underdetermined*.

Assuming uniformity. To tackle this problem, we start by assuming that SRC is a uniform overlay. Under this assumption, any two pixels in SRC will have the same RGBA values. As we will discuss later in this section, this assumption makes it possible to compute a candidate value for α starting from few pairs of pixels of DST and OUT; we can then use this candidate value for α to compute a candidate value for each pixel of SRC. Once we have computed SRC, we can then verify whether our initial assumption was correct: is the resulting SRC a semi-transparent uniform overlay? If the answer is affirmative, we know that OUT was computed from DST by applying a uniform "delta," and we consider this scenario as benign. If the resulting SRC is not uniform, there are two possible scenarios. In the first scenario, the "real" SRC was in fact uniform, but our candidate for α was wrong: in this case we raise an unnecessary warning. In the second scenario, the "real" SRC was not uniform. In this case, our procedure does not guarantee us to recover the "real" values for SRC. However, for our analysis, this is acceptable: if we have determined that SRC is not uniform, we can already flag the overlay as suspicious, without needing to know the real values of SRC.

α and SRC computation. Even under the assumption that SRC is a uniform overlay, it is still not possible to calculate RGBA values starting from one pixel only. However, it is possible to do so when considering *pairs of pixels*. Consider a pair of pixels for DST and OUT (two of each). We can write two equations for each of its RGB component (thus six in total), and we have 4 variables (SRC's

RGBA values). For example, if we consider the red component, the equations are:

$$\begin{cases} OUT_1^{red} = round(SRC^{red} * \alpha + DST_1^{red} * (1 - \alpha)) \\ OUT_2^{red} = round(SRC^{red} * \alpha + DST_2^{red} * (1 - \alpha)) \end{cases}$$

Under the assumption of a uniform SRC, we can compute a candidate value for α:

$$\hat{\alpha} = 1 - \frac{OUT_1^{red} - OUT_2^{red}}{DST_1^{red} - DST_2^{red}}$$

We note that due to rounding errors, the computed value is not precise. It is however possible to calculate a bound (all the calculation steps are reported in Appendix A), which results to be:

$$\hat{\alpha} - \frac{1}{DST_1^{red} - DST_2^{red}} \leq \alpha \leq \hat{\alpha} + \frac{1}{DST_1^{red} - DST_2^{red}}$$

That is, the smaller the difference of values for two of DST pixels, the greater the error is. To address these errors, in practice, we select multiple pairs of pixels, so that the bound is refined until a pre-defined precision is met (5/255 in our implementation). Once a candidate value for α is computed, we can then use it to calculate values for all pixels of SRC (by using the standard alpha blending equations: now we have three variables and three equations).

α and SRC analysis. At this point, we have a candidate value for α and the RGB values for SRC. Again, the value of α (and thus the values for SRC) are obtained under the assumption that SRC is indeed uniform. Now we have what is necessary to verify our assumption and whether SRC is a benign overlay. First, we verify that the value of α is lower than a certain threshold (0.94 in our implementation) to ensure it is indeed non-opaque. SRC is then analyzed for uniformity. To this end, we calculate a uniformity score γ that captures the range of different colors that are present in SRC, defined as

$$\gamma = \sum_{color \in \{red, green, blue\}} (\max_{p \in SRC} p^{color} - \min_{p \in SRC} p^{color})$$

In words, we consider the sum of differences between the max and min values for the red, green, and blue color components. This score is very low for uniform overlays, while it is much higher for non-uniform ones. Note that we opted for this simple metric instead of more complex ones like variance deviations because these latter ones change depending on the size of the image, and it is easier for an attacker to "hide" their pixel modifications within a low variance. For our experiments, the threshold we selected to discriminate between benign and suspicious cases is 100. This threshold was selected empirically: we refer the reader to Section 8.2 and Figure 4 for a discussion.

The full algorithm. To ease our explanation, we have omitted a number of additional aspects and optimizations, which we now describe. When we compute the candidate value for α, there are some corner cases that offer the possibility for optimization or simpler procedures. If $\alpha = 1$ (or, more in general, a value higher than our threshold), we flag SRC as suspicious, without even computing it: in fact, no matter what the values are, we would not have a semi-transparent overlay, which we do not want to allow. If $\alpha = 0$, then it means that the two DST and OUT pixels are the same, which implies that SRC would be a fully transparent overlay. In this case, instead of explicitly computing SRC, we perform a pixel-by-pixel comparison between DST and OUT to determine whether SRC is in fact fully transparent. If not, we flag the overlay as suspicious. This allows us to detect pixel-level modifications, such as a subtle change of price of an app. It is also possible that, while considering multiple pairs of pixels, we would encounter a situation where incompatible candidates for α are extracted. This implies that the SRC is not uniform (because α is not always the same) and we thus flag the overlay as suspicious. We also note that since we focus on the "central" part of the screen, these image analysis steps process a cropped version of DST and OUT. (In our current implementation, starting from a 720×1280 image, we discard 120 pixels from the left and right margins, and 200 from the top and bottom. We empirically selected these parameters taking into account the insights discussed in Section 5.)

The last corner case relates to the fact that a good candidate for α can be determined only when two pixels of DST are different enough. If DST is particularly uniform, it could be difficult to find those pixels randomly. For this reason, if after a number of attempts (100, in our implementation) we cannot find pixels that are suitable, our analysis does a full scan of DST to find such pixels. Unless DST is a particularly uniform image where all pixels are the very similar (a scenario that does not happen in practice for any useful app), we raise a warning. We note that pixel-picking may generate unneeded warnings in the worst case, but there is no risk of having attacks going unnoticed.

Performance considerations. We envision this system to be used only when a number of necessary conditions for clickjacking attacks are satisfied: the user clicked on a sensitive button and the central part of the screen is covered by one or more overlays. This is a rare condition in practice that only users of screen filters apps or users under attack would encounter. Nonetheless, the performance impact is negligible (see Section 8.2). In the worst case, the analysis needs to do two scans of the images: the first one to compute a candidate value for α (in the corner case mentioned above), the second one to compute and analyze SRC (these two steps can be done in-line). We notice that the first condition is so rare in practice that it was never triggered in our test cases, even in the simulated ones that are mostly uniform screens. Thus, in a real-world scenario, where apps do rarely display completely uniform screens, the expected worst-case is to perform one full pass on the images. The best case is to compute a value of α that does not require the computation of SRC to make a determination (e.g., $\alpha = 1$). Finally, we note that the performance of this analysis pass does not depend on the number or typology of overlays in the system, as we work at the image level.

7.3 Implementation and Other Aspects

We have implemented the core deblending algorithm in Python (for experimenting with the different thresholds and accuracy tests), and in C (for the performance tests on a real mobile device). As we will discuss in the Section 8.2, even if our implementation is single-threaded, the performance impact is negligible (especially when considering that this algorithm enters into play only when specific conditions are met, i.e., the user clicked on a sensitive button and the central part of the screen is covered by one or more

overlays). We would like to stress that this is just a prototype, and that actual implementation of this system could take advantage of the device's GPUs, making this analysis step even faster and less invasive. We have also implemented a modification to the Android framework to expose a new API that allows the system to extract both DST and OUT. This is implemented by modifying the framework with a special branch so that, if certain conditions apply, only layers associated to a given app (identified by its package name) are passed to the renderer, so that DST can be properly computed. Again, what we have implemented is a research prototype—an actual implementation of this system would simply store the partial results of the rendering in optimized data structures and make use of the GPU.

Another important aspect is handling race conditions: from the conceptual point of view, an attacker could deceive the user and hide the malicious overlays just before the user's click. This is a known problem that has been addressed in previous works [10, 21, 22]: if a sudden change is detected upon a click on a sensitive button, then the situation should be treated as suspicious no matter how the overlays look like.

8 EVALUATION

This section describes ClickBench, a benchmark tailored to specifically evaluate clickjacking solutions, and then discusses the accuracy, robustness, and performance of our prototype.

8.1 ClickBench

In an attempt to make this research area more systematic, we built the first comprehensive benchmark, called ClickBench, to evaluate clickjacking approaches. It is in fact easy to miss protection against specific corner cases, or to develop secure proposal that would interfere with existing apps [10, 17, 29]. Our dataset includes a total of 104 test cases. Each test case is constituted by pairs of images: the "pure" view of a target app that needs to be protected from clickjacking (i.e., DST), and what the user actually sees on the screen (i.e., OUT). The test cases were selected with the specific intent of representing the various scenarios we have encountered throughout our study. Among these 104 test cases, 47 represent benign use cases, while 57 represent malicious scenarios. We will release ClickBench to the community.

Benign samples. This set is constituted by 47 samples, and it aims at representing real-world benign use cases that rely, in one way or another, on the creation of persistent overlays (i.e., overlays that are not briefly shown just for the sake of a quick notification: these scenarios, in fact, would not create backward compatibility issues). 30 of them are obtained by running 10 real-world screen filter apps (like the ones listed in Table 1) each of which with three different settings (e.g., the tonality of the filter, the darkness, and the transparency). In all these cases, the overlays are drawn fullscreen, semi-transparent, and passthrough. We chose the remaining 17 samples with the intent of covering all the various scenarios described in Section 5.2: 10 create floating widgets (1 VPN & networking app, 1 productivity app, 2 audio and video players, 1 audio and video editor, 2 apps from status notification category, 2 custom launcher and 1 communication app); the last 7 test cases were built by using a combination of screenfilter applications and floating widgets.

Malicious samples. This set is constituted by 57 test cases. 10 of these samples aim at representing all known techniques and attack scenarios described in the literature [2, 10, 17, 24, 26, 29–31]. Some of these works and posts are overlapping, but we made sure to cover the two known classes of clickjacking techniques: traditional "clickjacking" and "context-hiding" attacks. To the best of our knowledge, all known attack scenarios are properly represented in ClickBench. The remaining 47 samples have been specifically developed to stress-test our own proposal by simulating different scenarios of the aforementioned attacks. For example, one of these examples aims at simulating a subtle attack that only modifies few pixels on the screen to change a price displayed on the Play Store (see Figure 5a and 5b). We also had a chance to externally validate our work with a *never-seen-before real-world malware*: While writing this paper, MWR labs published a blog post where they documented how they could mount a context-hiding attack (and perform clickjacking) against the Android SystemUI pop-up used to display the RSA authentication prompt when an adb server attempts to connect [15]. We took this opportunity to test ClickShield against this test case, which we were not aware of before finalizing the approach and the various thresholds, and we added this sample to ClickBench.

8.2 Evaluation

We now describe the accuracy results when running our system against ClickBench.

Accuracy. When run against our benchmark dataset, our system was able to detect all malicious cases as possible attacks (including the sample we had never seen), and it flagged all benign test cases as not problematic, except one. The benign test case that is flagged as potentially problematic is an app belonging to the F-Droid dataset (see Section 5.2), and it creates an opaque widget in the middle of the screen to preview a video. Figure **??** in the Appendix shows this app in action.

Although this is a benign scenario, we believe our prototype "correctly" flags it as potentially problematic: this overlay is in fact covering an important portion of the screen and there is a chance that the user may be deceived—this is *exactly* what an attacker could do. We note that, in a real-world deployment, this scenario generates an unnecessary warning only when the user is watching the preview and, at the same time, she is interacting with a security-sensitive app: if such condition is verified, ClickShield would then automatically make the overlay disappear and would ask the user to click again. Since we have not found similar samples in popular apps on the Play Store (this sample belongs to the 15 of the F-Droid dataset), since a false warning would be raised only in very narrow circumstances and do not cause significant UI interference, and since there is an actual risk that the user could be deceived (few pixels modification are enough), we believe this is an acceptable trade-off. We reiterate that all the popular screen filters, apps creating widgets, and the combination thereof were correctly classified as not problematic.

Robustness. For 47 of these cases, the candidate α is computed to be zero, and thus our system performs a pixel-by-pixel comparison. For 44 of these cases, ClickShield detected opaque overlays in the central area of the screen, and thus flagged them as malicious. In all

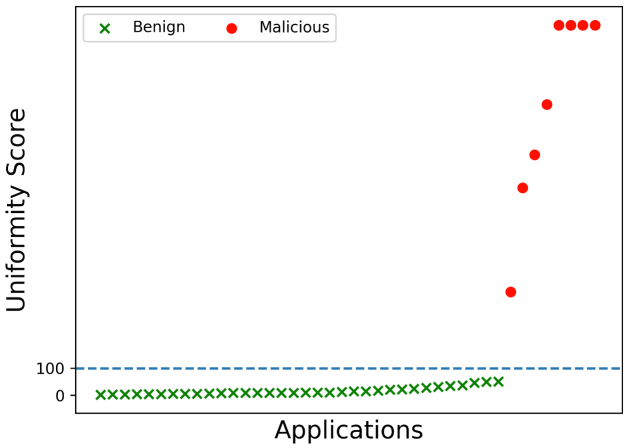


Figure 4: This plot shows the distribution of the uniformity score across our benchmark dataset. The plot highlights how there is a clear cut between the benign and the malicious test cases. 100 is the score threshold we used in our prototype. We note that this plot only concerns the 42 test cases for which the uniformity score needed to be computed.

benign cases, ClickShield was able to reconstruct the correct SRC. For these 44 cases, ClickShield needed to sample an average of 6.13 pairs of pixels to compute a good estimation for α (minimum of 1 and maximum of 62, which was needed for the most complex of our simulated malicious examples). The full SRC analysis is triggered in 42 of these test cases (in 2 of the 44 cases, α was above the maximum opaqueness threshold, and SRC was not needed). Finally, Figure 4 shows the distribution of the uniformity scores γ obtained for these test cases. The figure shows how the benign and malicious test cases are clearly separated. This implies that our system is not very sensitive to the specific threshold we are using.

We note that some test cases have been developed specifically to break our own defense. For example, one of them consists of a semi-transparent overlay with a few pixels modified so that, when overlapped with the Play Store, the price of an app would change from 2.99 to 1.99. The semi-transparent overlay is used to cause our system to detect $\alpha \neq 0$, and it thus triggers the SRC computation (which is the part of our work that is the most prone to noise and errors). The figures for these particular test cases are reported in Figure 5 and 6 in Appendix C. As the reader can see, the computed SRC is mostly uniform, but it does contain a few anomalous pixels, which are enough for our system to detect this scenario as suspicious.

Performance. We now discuss a performance evaluation for what concerns the major bottleneck of our approach, the deblender analysis. For this analysis, as we have mentioned, the worst-case scenario that one would encounter in practice is the need for a full scan of DST and OUT to compute and analyze SRC. We have run a number of experiments on a Google Nexus 5X, running the latest version of Android (8.0). We have written our analysis as a native code component. This component takes as input DST and OUT images, and it computes and analyze SRC, returning "benign" or "suspicious" as an outcome. We have performed this experiment 100 times and we have timed the computation and analysis of SRC. The mean of the execution time is of 12.73 *ms* (σ = 0.000456). We believe this overhead to be negligible, especially since this analysis step is performed only when specific conditions are met (i.e., the user clicked on a sensitive button and the central part of the screen is covered by one or more overlays). Moreover, we stress that for a real-world deployment of this system, one should take advantage of the GPU and other hardware optimizations. We believe the value of our work consists in showing that an efficient analysis technique to protect against clickjacking exists and that, according to our evaluation, it is effective.

9 THREATS TO VALIDITY

As for every protection mechanism, there is always the possibility that, when deployed in the real-world, there could be either backward compatibility issues or the possibility for evasion. This section discusses these risks and our efforts to minimize them.

Backward compatibility concerns. ClickShield raises warnings only when the central portion of the screen is covered by a non-uniform overlay *and* the user attempts to interact with a security-sensitive widget. Our survey (Section 5.2) appears to validate our assumption that these situations are very rare in benign scenarios. The risk of concrete usability issues is also mitigated by the fact that, if a warning occurs, ClickShield would simply remove the covering overlays and ask for an additional confirmation. One other concern is that our survey may not be representative. We addressed this concern by considering samples from different sources, including top apps on the Play Store and apps that we knew to be specifically problematic for clickjacking protections.

Evasion possibilities. One inherent assumption of our mechanism is that a successful attack needs to cover the central part of the screen. Once again, we relied on a survey (discussed in Section 5.1) to validate our hypothesis: in all cases the needed contextual information is prominently shown in the central part of the screen, and, at least for the 67 views covered by our survey, we are not aware of any technique to bypass our defense (which certainly already breaks all the known and new attacks presented in this paper).

One other concern is that our system shares one limitation with existing solutions (e.g., obscured flag): the developer needs to indicate which parts of the app are sensitive. We acknowledge that an ideal system would automatically address this aspect. However, this problem has been addressed in previous research [21], which could be used in conjunction with our system. That being said, our understanding is that many sensitive apps are left unprotected not due to an oversight, but due to the lack of a practical solution—the core problem that this work attempts to solve.

10 RELATED WORK

10.1 Offensive Research in Android UI

One research area in terms of attacks of Android UI relate to mobile phishing. A well-known form of UI attack on Android is phishing, also known as "task hijacking." For example, Rydstedt [23] demonstrated that mobile browsers are vulnerable to framing attacks, while several other works have shown how to lure users to enter

their credentials into malicious, spoofed UIs [4, 6, 8, 20]. The other class of UI attacks is clickjacking, the focus of this work. We have already mentioned several works on the topic from the research community [2, 10, 17, 29], and these techniques are finding their ways in real-world malware as well [24, 26, 30, 31].

10.2 Defenses against UI Attacks

A number of works have focus on defending from Android UI threats. The first group of works aims at defending from clickjacking attacks. One of the first defense proposal is by Niemietz et al. [17], which propose to add a security layer between overlays, so that no user input can pass across apps. While this would prevent clickjacking attacks, it would cause backward compatibility problems with existing apps (see Section 5.2). Another work in the area is [29], which proposes to detect malicious overlays considering a number of features: in their work, the authors consider as benign overlays that are either 1) semi-transparent or 2) clickable by breaking *observation* #3 and #4. As we have shown, these can be bypassed. Another recent proposal is [21], but it is bypassable via context-hiding attack [10]. Other works attempt to prevent UI attacks by detecting the presence of any overlay potentially obscuring the protected UI by providing the developer with a third-party library using obscure flag [25] or a11y [11]. The defense mechanism proposed in [10] is similar to "hide overlays" implemented by Google, and it shares the same backward compatibility concerns. Another defense proposal is [19], which proposed the Android Window Integrity policy: it makes sure the current visible window is not obscured by windows from another app. However, like the "hide overlay" mechanism it does not address the challenge of handling the many legitimate overlay use cases (see Section 5.2) and have to resort to whitelisting all legitimate apps that employ overlay windows, which is not scalable and might introduce security concerns. In fact, a malware could first display benign overlays, be added to the whitelist, and then perform unconstrained UI attacks. Our system instead, compared to [19, 21, 29], is different in multiple aspects: it does not rely on a whitelist approach and it is completely transparent to the user. Moreover, it evaluates the maliciousness of a given set of overlays at run-time, making the evasion techniques mentioned above, to the best of our knowledge, ineffective.

There are a number of other works that aim at defending from other classes of UI attacks. For example, [4, 9, 14, 19] aim at preventing phishing attacks on Android. Unfortunately, these works are ineffective against clickjacking attacks. It is important to denote how phishing is significantly different than clickjacking: in a phishing attack, the malicious application tries to lure the victim into inserting sensitive information like username and password by mimicking the legitimate application interface and behavior, while in clickjacking the user is lured to unknowingly interact with the (benign) target application beneath a malicious overlay. Finally, other works addressed the problem of automatically identifying usage of user sensitive input (e.g., user credentials) to automatically mark as "sensitive" and protect them [3, 12, 16]. These works do not tackle the problem of clickjacking, but they are very promising and complementary to our work.

11 CONCLUSIONS

In this work we have shown that clickjacking on mobile is still an open problem and that many first- and third-party apps are still affected, even if this class of attacks has been known for several years. After surveying how apps use overlays and explore the design space, we have proposed a new protection mechanism, ClickShield, based on image analysis technique, and we have showed its effectiveness by evaluating it against ClickBench. Unfortunately, even if we are disclosing the vulnerabilities we have discussed in this paper, it is unclear how the affected apps can prevent these attacks without additional support from the Android framework. We hope that Google will consider and implement our proposal, which we believe would allow apps to defense themselves from the threat of clickjacking, without risking public outcry due to backward compatibility issues.

ACKNOWLEDGEMENTS

This research was supported in part by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP006. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. We would also like to thank Anthony Trummer, Luca Carettoni, and, of course, Betty Sebright.

REFERENCES

[1] Yair Amit. 2016. 95.4 Percent of All Android Devices Are Susceptible to Accessibility Clickjacking Exploits. https://www.skycure.com/blog/95-4-android-devices-susceptible-accessibility-clickjacking-exploits/.
[2] Yair Amit. 2016. "Accessibility Clickjacking" – The Next Evolution in Android Malware that Impacts More Than 500 Million Devices. https://www.skycure.com/blog/accessibility-clickjacking/.
[3] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 2017. UiRef: analysis of sensitive user inputs in Android applications. In *WISEC*.
[4] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proc. of the IEEE Symposium on Security and Privacy*.
[5] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA*.
[6] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. 2014. Peeking Into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. of the USENIX Security Symposium*.
[7] F-Droid. 2018. Free and Open Source (FOSS) software on the Android platform. https://f-droid.org/en/
[8] Adrienne Porter Felt and David Wagner. 2011. Phishing on Mobile Devices. In *Proc. of IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*.
[9] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Proc. of Financial Cryptography and Data Security (FC)*.
[10] Yanick Fratantonio, Chenxiong Qian, Pak Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
[11] geeksonsecurity. 2018. Android Overlay Protector. https://geeksonsecurity.github.io/overlay-protector-website/.
[12] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *USENIX Security Symposium*.
[13] Google Inc. 2018. Documentation for the `FLAG_WINDOW_IS_OBSCURED` `flag`. https://developer.android.com/reference/android/view/MotionEvent.html#FLAG_WINDOW_IS_OBSCURED
[14] Luka Malisa, Kari Kostiainen, and Srdjan Capkun. 2015. Detecting Mobile Application Spoofing Attacks by Leveraging User Visual Similarity Perception. In *Cryptology ePrint Archive, Report 2015/709*.
[15] Amar Menezes. 2018. Privilege Escalation via adbd Misconfiguration. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-android-adbd-privilege-escalation-advisory-2018-01-17.pdf.

[16] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and Xiaofeng Wang. 2015. UIPicker: User-Input Privacy Identification in Mobile Applications. In *USENIX Security Symposium*.
[17] Marcus Niemietz and Jörg Schwenk. 2012. UI Redressing Attacks on Android devices. *Black Hat Abu Dhabi* (2012).
[18] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks Against Machine Learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*.
[19] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *Proc. of the Annual Symposium on Network and Distributed System Security (NDSS)*.
[20] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proc. of USENIX Security Symposium*.
[21] Talia Ringer, Dan Grossman, and Franziska Roesner. 2016. AUDACIOUS: User-Driven Access Control with Unmodified Operating Systems. In *Proc. of the Conference on Computer and Communications Security (CCS)*.
[22] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J Wang, and Crispin Cowan. 2012. User-driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proc. of the IEEE Symposium on Security and Privacy*.
[23] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. 2010. Framing Attacks on Smart Phones and Dumb Routers: Tap-jacking and Geo-localization Attacks. In *Proc. of the USENIX Conference on Offensive Technologies*.
[24] Tara Seals. 2016. Autorooting, Overlay Malware Are Rising Android Threats. http://www.infosecurity-magazine.com/news/autorooting-overlay-malware-are/.
[25] SFYLABS. [n. d.]. Client Side Detection (CSD). https://clientsidedetection.com.
[26] Tom Spring. 2016. SCOURGE OF ANDROID OVERLAY MALWARE ON RISE. https://threatpost.com/scourge-of-android-overlay-malware-on-rise/117720/.
[27] Cameron Summerson. 2017. How to Fix the "Screen Overlay Detected" Error on Android. https://www.howtogeek.com/271519/how-to-fix-the-screen-overlay-detected-error-on-android/
[28] Urbandroid Team. 2018. Twilight App. https://play.google.com/store/apps/details?id=com.urbandroid.lux&hl=en.
[29] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. 2016. Analysis of clickjacking attacks and an effective defense scheme for Android devices. *2016 IEEE Conference on Communications and Network Security (CNS)* (2016), 55–63.
[30] Martin Zhang. 2016. Android ransomware variant uses clickjacking to become device administrator. http://www.symantec.com/connect/blogs/android-ransomware-variant-uses-clickjacking-become-device-administrator.
[31] Wu Zhou, Linhai Song, Jens Monrad, Junyuan Zeng, and Jimmy Su. 2016. The Latest Android Overlay Malware Spreading via SMS Phishing in Europe. https://www.fireeye.com/blog/threat-research/2016/06/latest-android-overlay-malware-spreading-in-europe.html.

APPENDIX

A BOUND CALCULATION

Here we present the steps to calculate the bound for α. We assume that SRC is a uniform overlay. Thus, SRC's RGB and α are assumed to be constant. Under this assumption, we want to compute a candidate value for α. Let's consider one random pair of pixels, (p_1, p_2). Each of these pixel has three components, RGB. For each color component, we can write a pair of equations. These equations are directly derived from the alpha blending formulas:

$$\begin{cases} OUT_1^{red} = round(SRC^{red} * \alpha + DST_1^{red} * (1-\alpha)) \\ OUT_2^{red} = round(SRC^{red} * \alpha + DST_2^{red} * (1-\alpha)) \end{cases}$$

where OUT_i^{red} and DST_i^{red} indicate the red component of i-th pixel of OUT and DST, SRC^{red} and α indicate the red and alpha component of SRC (under our assumption, all pixels are the same). α is a value between 0 and 1 (while all the other ones are an integer from 0 to 255).

Now, the problem is the *round* operation, which makes us lose information. This means that the OUT_i^{red} values we have, are not the "real ones," but a close approximation. We can rewrite the equations above by making the rounding error explicit:

$$\begin{cases} OUT_1^{red} = SRC^{red} * \alpha + DST_1^{red} * (1-\alpha) + \epsilon_1 \\ OUT_2^{red} = SRC^{red} * \alpha + DST_2^{red} * (1-\alpha) + \epsilon_2 \end{cases}$$

where ϵ_1 and ϵ_2 model the rounding errors. Given the nature of the errors, we know that

$$-\frac{1}{2} \leq \epsilon_i \leq \frac{1}{2}$$

Now, let us solve the system of equations for α. By subtracting the second equations from the first one, we obtain:

$$\alpha = 1 - \frac{OUT_1^{red} - OUT_2^{red}}{DST_1^{red} - DST_2^{red}} - \frac{\epsilon_1 - \epsilon_2}{DST_1^{red} - DST_2^{red}}$$

Now, let us indicate with $\hat{\alpha}$ the first part of the equation, and with $\Delta = \epsilon_1 - \epsilon_2$. We have:

$$\alpha = \hat{\alpha} - \frac{\Delta}{DST_1^{red} - DST_2^{red}}$$

Given the constraints on ϵ_i, we have the following constraint on Δ:

$$-1 \leq \Delta \leq 1 \qquad or \qquad |\Delta| \leq 1$$

which implies that

$$\hat{\alpha} - \frac{1}{DST_1^{red} - DST_2^{red}} \leq \alpha \leq \hat{\alpha} + \frac{1}{DST_1^{red} - DST_2^{red}}$$

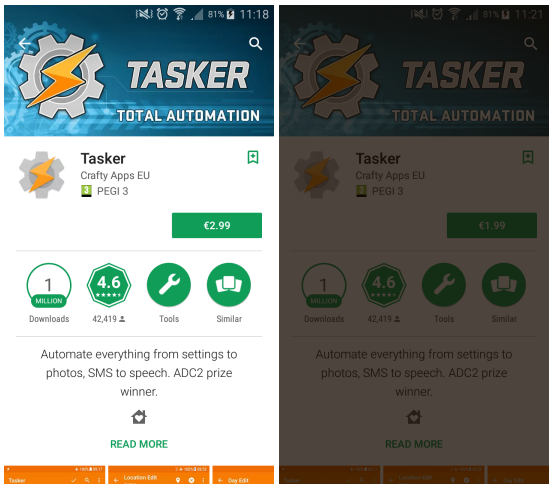
This is our bound on $\hat{\alpha}$. Note that we can compute a candidate value for $\hat{\alpha}$ for each of the RGB components of each pairs of pixels. Of course, some pixels will give us a less information than others, but we can always select more pixels to refine the bound we have.

B SCREENFILTERS

Package Name	# of Installs
jp.ne.hardyinfinity.bluelightfilter.free	10M-50M
com.urbandroid.lux	5M-10M
com.eyefilter.nightmode.bluelightfilter	5M-10M
pt.bbarao.nightmode	5M-10M
com.eyefilter.night	5M-10M
com.arrowsapp.nightscreen	1M-5M
es.richardsolano.filter	1M-5M
com.mlhg.screenfilter	500K-1M
bluelight.filter.sleep.warmlight.eyes.battery	500K-1M
com.ascendik.eyeshield	100K-500K
com.kapron.ap.eyecare	100K-500K

Table 1: These apps' core functionality is to create a screen filter to change its color tonality. We have manually tested all these apps, and they all require the "draw on top" permission, and they all work by creating a persisent, fullscreen, semi-trasparent overlay on top of every other app. Thus, these apps directly conflict with all existing protection mechanisms against clickjacking, causing backward compatibility issues.

C ADDITIONAL IMAGES

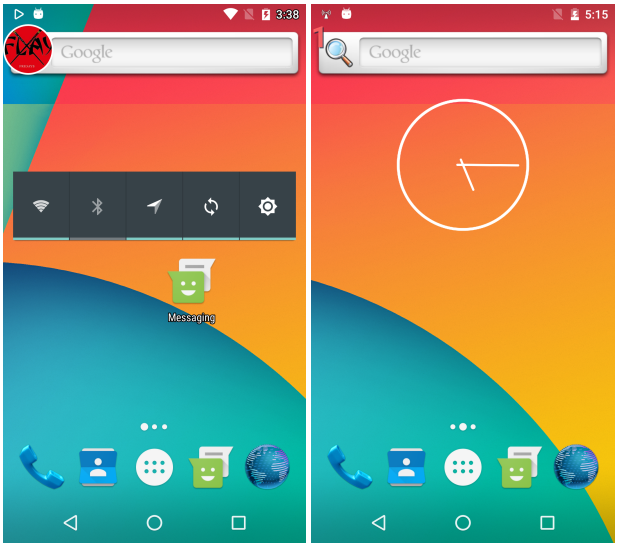


(a) (b)

Figure 5: (a) The original non-modified Play Store, where displayed price is 2.99 (DST). (b) A simulation of an attacker attempting to bypass our defense mechanism by creating a semi-transparent overlay (which triggers α and SRC analysis) and where the price has been subtly changed from 2.99 to 1.99 (OUT).

(a) (b)

Figure 6: (a) This shows the candidate version for SRC computed by ClickShield. (b) This shows the same SRC image, but "zoomed in" to show the small pixel difference that responsible for the subtle price change. This scenario represents one of the most challenging scenarios for our defense mechanism, but it is properly handled: the small difference is already enough to be flagged as suspicious, and alert the user.



(a) (b)

Figure 7: Application drawing widget shown in (a) is using a small opaque widget drawn at margin: it is configured to capture the click of the user. When the user press on top of the widget, it will show some information about the current audio being played. The application drawing (b) instead is using a different configuration of the widget: this time it is not configured to intercept user click. The widget is drawn on the upper-left corner of the screen and the number "1" represents how many available wifi network are present nearby.

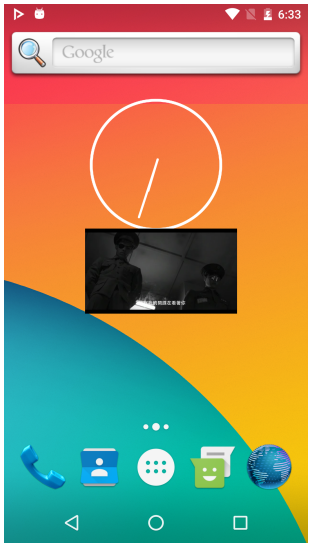
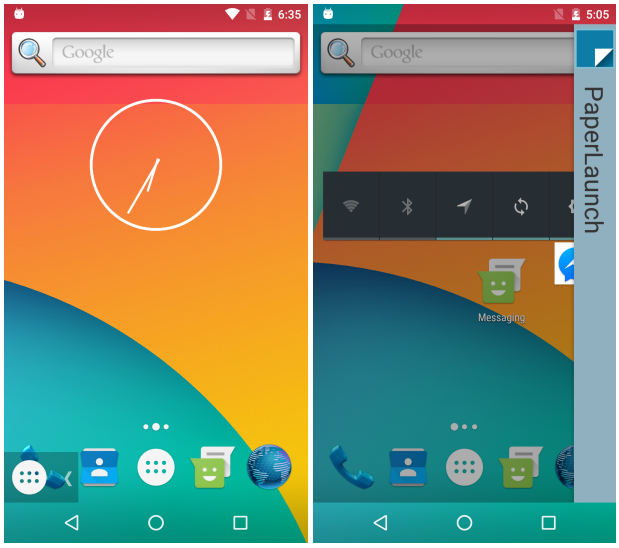
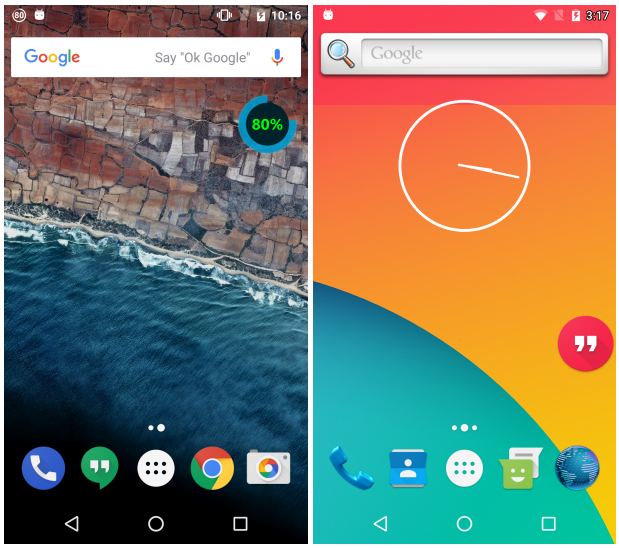


Figure 8: This represents the scenario for which ClickShield raised a false warning.



(a) (b)

Figure 9: Both the samples are representing a "custom launcher." (a) is drawn in the bottom-left corner and it is configured as a small widget, opaque and clickable overlay: once the user interacts with it, the widget will show some applications to launch. Similarly (b) is providing the same functionality but the way the widget is configured is different: for this sample, the widget is drawn in the right side of the screen as an opaque and clickable overlay.



(a) (b)

Figure 10: (a) shows a custom floating, semi-transparent and clickable widget drawn by a third-party application: it is used to better inform the user about the status of the battery. (b) represents a similar widget as the one described before (this time it is fully opaque) with a different functionality: once clicked, the widget will redirect the user to a "note" she wrote (like a shortcut).

D SYSTEMATIZATION OF CLICKJACKING ATTACKS

Vulnerable Component	Status	Already known?	# of clicks	Net effect
Permissions	Fixed via Hide Overlays	Yes [17]	1	Attacker can lure the user to grant additional permissions
Accessibility Service (a11y)	Fixed via Hide Overlays	Yes [2, 10, 17]	3	Attacker can lure the user to grant a11y permission
Camera	Unfixed	Yes [29]	1	Attacker is able to capture image from camera
Contact	Unfixed	Yes [29]	1	Attacker is able to get access to contacts
SoundRecorder	Unfixed	Yes [29]	1	Attacker is able to record sound
Text Messages	Unfixed	Yes [29]	1	Attacker can send SMS on behalf of the victim
Package Installer	Fixed via Obscure Flag	Yes [29]	1	Attacker is able to install and uninstall arbitrary application
Google Play Store	Unfixed	**No**	2/3	Attacker can install and open an arbitrary app from the Play Store with permissions enabled at install-time
Chrome Web Browser	Unfixed	**No**	Variable	Attacker can perform web clickjacking attacks and bypass web-related defense mechanisms
GMail Client	Unfixed	**No**	1	Attacker can send emails on behalf of the victim
Facebook and Twitter	Unfixed	**No**	1	Attacker can impersonate the user on social networks
WhatsApp and Signal	Unfixed	**No**	1/2	Attacker can deanonymize the victim by leaking her phone number or send messages on her behalf
Google Authenticator	Unfixed	**No**	1 (long click)	Copy the GAuth token to the clipboard, which is readable by any app (no permission required)
Lookout Mobile Security	Unfixed	**No**	3	Attacker can silently disable the security checks

Table 2: Systematization of the attacks. For each attack, we report the vulnerable component, the current status of the patching, if the attack was already known, the number of clicks to hijack, and what is the net effect of the attack.